Exploiting Iteration-Level Parallelism in Dataflow Programs

Lubomir Bic
John M. A. Roy
Mark Nagel

Department of Information and Computer Science
University of California, Irvine, CA 92717

Technical Report 91-57

Exploiting Iteration-Level Parallelism in Dataflow Programs †

Lubomir Bic
John M. A. Roy
Mark Nagel

Department of Information and Computer Science
University of California, Irvine, CA 92717

ABSTRACT

The term "dataflow" generally encompasses three distinct aspects of computation—a data-driven model of computation, a functional/declarative programming language, and a special-purpose multiprocessor architecture. In this paper we decouple the language and architecture issues by demonstrating that declarative programming is a suitable vehicle for the programming of *conventional* distributed-memory multiprocessors.

This is achieved by appling several transformations to the compiled declarative program to achieve iteration-level (rather than instruction-level) parallelism. The transformations first group individual instructions into sequential light-weight processes, and then insert primitives to: (1) cause array allocation to be distributed over multiple processors, (2) cause computation to follow the data distribution by inserting an index filtering mechanism into a given loop and spawning a copy of it on all PEs; the filter causes each instance of that loop to operate on a different subrange of the index variable.

The underlying model of computation is a dataflow/von Neumann hybrid in that exection within a process is control-driven while the creation, blocking, and activation of processes is data-driven.

The performance of this process-oriented dataflow system (PODS) is demonstrated using the hydrodynamics simulation benchmark called SIMPLE, where a 19-fold speedup on a 32-processor architecture has been achieved.

Keywords and Phrases: dataflow, declarative programming, implicit parallelism, distributed-memory multiprocessors

† This work was supported by the NSF Grant CCR-8709817.

Contents

1. Introduction

The programming of parallel computer systems is a difficult and highly error-prone task. This is due primarily to the lack of adequate facilities to describe a problem to a parallel machine at a high level, without sacrificing performance. The current state of the art in programming parallel machines efficiently is to let the programmer explicitly partition the program into processes and insert the necessary synchronization and communication primitives.

There are several schools of thought on how to make parallel processing more accessible and more effective, as shown graphically in Figure 1. The most common (and least revolutionary) approach is to rely on existing sequential languages. These are either extended to allow the programmer to express parallelism explicitly, or sophisticated compilers capable of extracting parallelism from a given program automatically are employed.

At the opposite end of the spectrum are approaches which completely abandon conventional von Neumann systems in favor of radically new languages and architectures. Perhaps the best known representatives of this approach are dataflow systems, which start with a data-driven model of computation, employ a functional/declarative style of high-level programming, and design special-purpose architectures targeted specifically to the execution of dataflow programs.

Our approach is intermediate to the above two extremes. We concentrate on conventional multiprocessors and investigate how to effectively program them using new languages. Specifically, our goal is to demonstrate that a declarative language, intended primarily for the programming of special-purpose dataflow architectures, is a highly-suitable vehicle for the programming of commercially available multiprocessors.

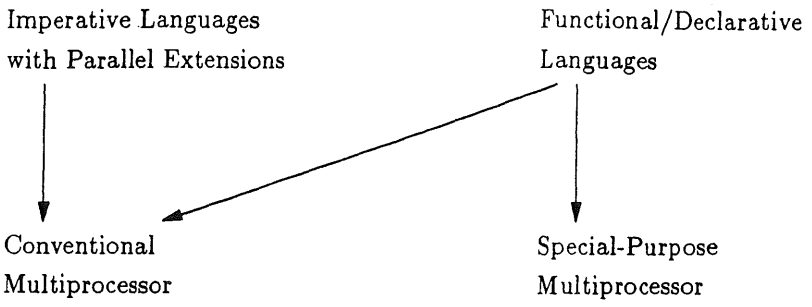

Figure 1

Approaches to Parallel processing

The paper is organized as follows. Section 2 motivates the use of declarative programming. Sections 3 and 4 described the principles of PODS — the Processes-Oriented Dataflow System. Specifically, Section 3 describes how dataflow programs are transformed into communicating processes while Section 4 presents the mechanisms for distributing their execution over multiple processors. Section 5 then presents the results of the simulations. Finally, conclusions and comparisons to other approaches are given in Section 6.

2. Declarative Programming

Most programming languages today are based on the imperative style of programming, where data is viewed as passive elements in storage, which are accessed and manipulated by a stream of instructions under the control of a program counter. Alternative programming styles, including functional, logic, or object-oriented, have been developed in the past, with the objective of facilitating the task of program development. Specifically, functional programming has been studied extensively in the context of parallel machines, due to their clean mathematical properties and the lack of side-effects. Unfortunately, the syntax of pure functional languages, such as FP [BAC78], where programs are essentially compositions of nested functions, is considered too "user-unfriendly" for the development of large

scientific or commercial programs. They also lack the support for large data structures, such as arrays and matrices, which are considered essential for scientific programming.

In the context of dataflow systems, several functional languages, which overcome the above problems, have been developed. The best known such languages are Val [ACK79], Sisal [McG85], and Id [NIK88], which all support the common control and data-manipulation constructs found in imperative languages, including if-then-else statements, for- and while-loops, procedure calls, and various facilities to manipulate data structures and streams. This makes the development of large programs using these languages not only possible but even easier than using conventional languages, such as C or Fortran [ARV88].

In our research we use Id, which is a functional language augmented with a parallel data-structuring mechanism called I-structures [ARV89]. I-structures may be viewed as arrays that obey the principle of *single assignment*, which is at the core of all functional languages. This principle states that any element of the array may be written into only once. After it has been written, it may be read any number times. The necessary synchronization, which delays all read requests until a value has been written and which also reports any attempts to rewrite a value as a single-assignment violation, is enforced automatically by the I-structure memory.

A *declarative* style of programming is defined as functional programming, augmented with the concept of single-assignment arrays, as embodied in the latest version of Id, called Id Nouveau [NIK87, ARV87A]. The main difference that sets declarative programming apart from pure functional programming is that *referential transparency* is given up. That is, values returned by two calls to the same function with the same arguments will not necessarily be indistinguishable. The advantage is that one can alter a data structure once it has been created, instead of having to specify the contents of all its elements at creation time, as is

demanded by a purely functional language. This allows a style of programming which is more atune with the way programmers think about a problem. Specifically, an I-structure may be defined initially as an empty set of slots, which may be filled (and consumed) later by subsequent computations. This is very similar to using arrays in conventional languages, except for the single-assignment property.

Note, however, that the *Church-Rosser property* [LAN65], also called the confluence property, is preserved by declarative programs. This requires that the answer computed by an expression be unaffected by the choice of which subexpressions are evaluated first. Since I-structures enqueue all early reads until the element is written, and each element preserves single assignment, I-structures preserve the Church-Rosser property. No matter how one interleaves the execution of reads and writes, every fetch to a given I-structures element always returns the same value. Hence the overall program determinacy is guaranteed even if the machine exhibits non-determinacy in instruction scheduling.

3. Subcompact Processes

For the purposes of this paper, it is not necessary to understand the exact syntax or semantics of Id. (The interested reader is referred to [NIK87, ARV87A, NIK88].) It is, however, important to point out that the underlying clean semantics of Id (and other functional/declarative languages) make it possible to translate high-level language programs into dataflow graphs, which precisely capture the data dependencies among all operators.

The following is a simple Id program which fills a 2-dimensional array A by computing a value for each of its elements.

```
A = matrix(50,10);
for i = 1 to 50
  for i = 1 to 10
    A[i,j] = f(i,j);
```

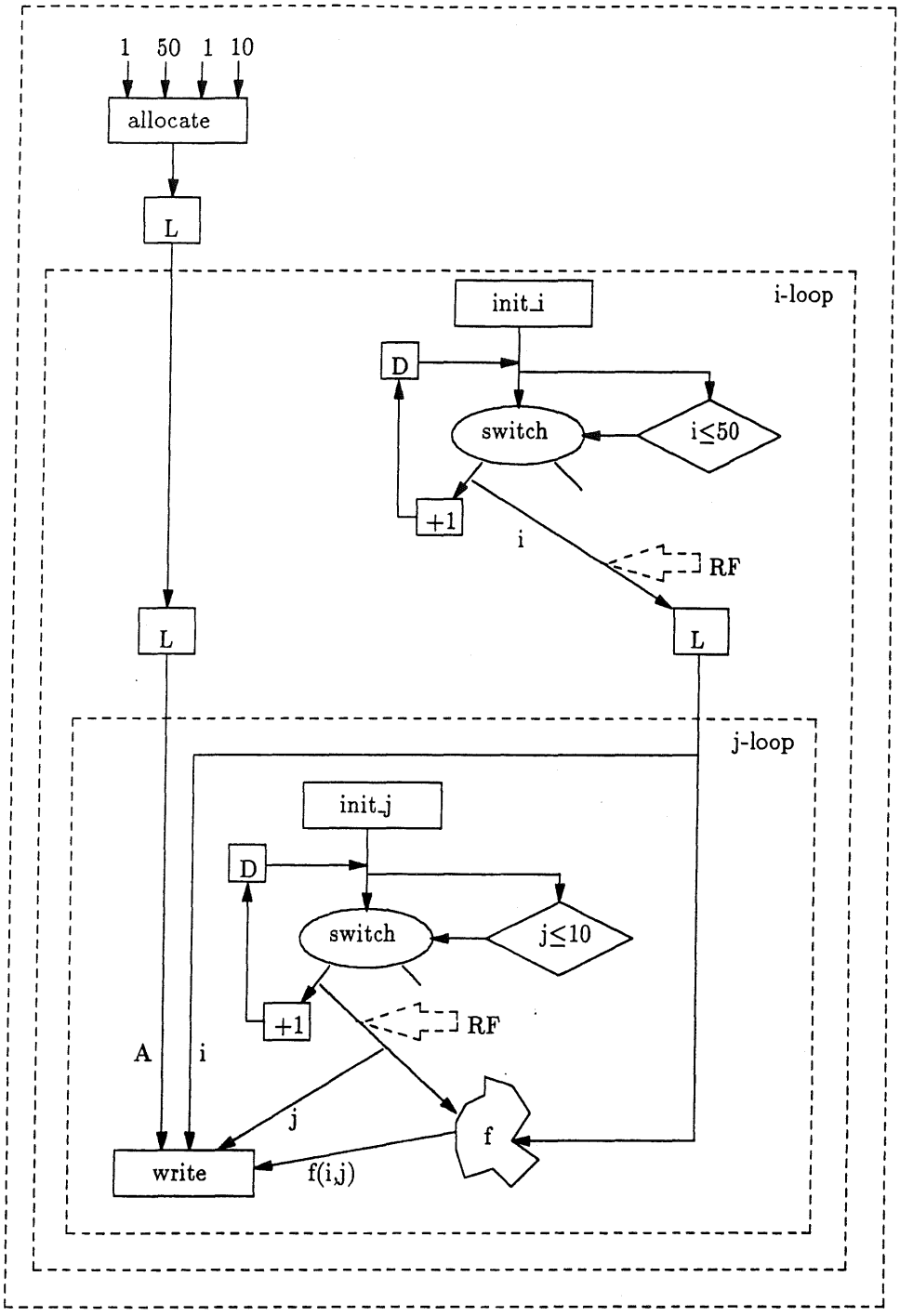

Figure 2

Example of a Dataflow Graph

Figure 2 shows the essential parts of the corresponding dataflow graph. There are three separate scopes (code blocks), each entered through L operators (or, initially, a function call) which create a new context for the corresponding scope. The outer-most scope causes the space for the array A to be allocated, given the array bounds. The array ID is passed into the inner-most loop where it is used by the write operator.

The middle scope contains the code to generate the index sequence for i. The initial value for i (1, in this case) is circulated through the switch, increment, and D operators until it reaches the value 50. A copy of each i is passed into the inner context, where the index sequences for j are generated in a similar manner and, together with each i, are used to compute the function f and to write the value into the appropriate element of A. (The meaning of the dashed arrows labeled "RF" will be explained later.)

The dataflow graph captures the underlying data-driven semantics of the high-level program. There is no program counter. Instead, each node of the dataflow graph is an autonomous instruction, capable of executing whenever it receives all its operands along the graph arcs. In principle, each node may be viewed as an independent "process", which is instantiated when the necessary operands arrive, performs the prescribed operation, and sends the resulting values to other such "processes" that need the data as their inputs.

One way to exploit this highly asynchronous model of computation is to build special-purpose dataflow architectures, capable of tolerating the resulting overhead of instruction-level parallelism. Another approach, the one taken in our project, is to increase the level of granularity from a single instruction to a group of instructions, thus permitting them to execute as threads or light-weight processes. In our case, the objective is to execute such processes on a conventional

multiprocessor. Due to the minimal state associated with each such processes, we refer to them as Subcompact Processes (SPs)†.

The idea of forming sequential threads of computation out of dataflow programs is the essence of dataflow/von Neumann hybrids, a number of which have been proposed in recent years [CUL91, GAO90, GAU90, GRA89, IAN88, NIK89, ROG89, SAK89]. One of the main issues, which differentiates the various approaches, is how to create processes, i.e., how to subdivide the original dataflow graph or program into sequential code segments. Our initial approach [BIC90] was to divide the graph into paths according to their data-dependencies using a depth-first coloring scheme. Unfortunately, the resulting SPs were too small to execute efficiently on conventional distributed memory multiprocessors. The current approach is to make each function a new SP and, within each function, make each loop iteration level a separate SP. This coincides with the subdivision of the code generated by the Id Nouveau compiler. Hence each code block, when invoked, becomes a separate SP.

Consider again the dataflow graph in Figure 2. Each of the three scopes would be instantiated as an SP. In particular, SP1 performs the array allocation and then invokes SP2, which generates the 50 index values for i. For each i, it invokes SP3, which generates the 10 j-indices and performs the corresponding computation for each. In addition, each individual SP can be distributed over all PEs where each copy operates over a distinct subrange of the array.

While the execution within a given SP is control-driven, the instantiation and activation of SPs is triggered by the arrival of operands (i.e., still data-driven). An SP is passive as long as its first instruction is disabled. When all operands for the first instruction have arrived, the SP becomes active. This is accomplished by loading the SP into execution memory and creating a simple process control block (PCB) for it, consisting essentially of the starting address of the SP in execution

† We have borrowed the term "subcompact" from the automobile industry to mean having the smallest possible state to still justify calling it a process.

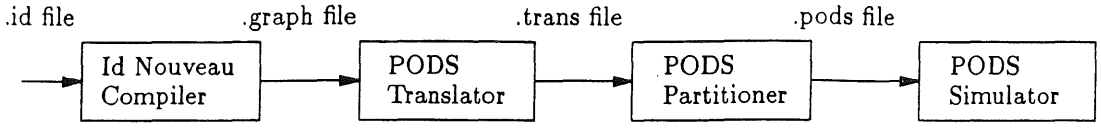

Figure 3

Organization of PODS

memory, a program counter pointing to the current instruction, and a status field indicating whether the process is running, ready, or blocked.

The three states are analogous to those found in most operating systems. An SP is running until it reaches the end of the SP (at which time it is destroyed) or until it encounters an instruction that does not yet have all its operands present. In the latter case, the SP is blocked and the PE switches to another ready SP. The blocked SP changes its status to ready as soon as the last operand for the current instruction arrives. This process-oriented viewpoint permits us to execute a dataflow program as a collection of communicating SPs. Hence we refer to our system as PODS — a *Process-Oriented Dataflow System.*

The overall organization of PODS is depicted in Figure 3. At the top level is the Id Nouveau compiler developed at MIT, which translates Id source programs into dataflow graphs. These are then translated in two steps into code executable on the PODS simulator. The first step, performed by the PODS Translator, converts code blocks into SPs. This consists primary of (1) eliminating synchronization instructions used to implement k-bounded loops [ARV86A, CUL89], and (2) ordering instructions within each code block according to dataflow arcs such that no instruction depends on data generated by an instruction lower in the sequence. An important implicit change in semantics also takes place. The instructions within a code block are now viewed as a single sequential SP. This implies that every instruction, in addition to generating data values for subsequent instructions, must

also modify a program counted. In most cases, the program counter is simply incremented. In the case of a switch operator, the program counter is either incremented, thus pointing to the true branch of the statement, or set to a new value to skip over to the false branch.

The next step, performed by the PODS Partitioner, is to modify the SPs to achieve distribution. This is accomplished by inserting three types of primitive into the SP code: a *distributing allocate operator*, which causes arrays to be spread over different PEs, a *distributing L operator*, which causes a given SP to be spawned on multiple PEs concurrently, and a *range filter*, which guarantees that each of the replicated SPs operates on a different index range. These primitives form the core of PODS and are the major focus of the next section.

4. Distributed Execution

PODS supports both functional and data parallelism. The main focus of this paper, however, is on data-parallelism. Specifically, we are interested in iteration-level parallelism (rather than instruction-level parallelism), which plays a prominent role in most scientific computations. The goal is to distribute the iteration space of a loop and the corresponding data over multiple PEs such that each would operate on a different subrange of the index space.

PODS has been targeted to a distributed memory MIMD architecture with a *non-uniform* access to memory, such as the Intel iPSC/2, the n-cube, or the cosmic cube. This means that access to non-local memory is much slower (involving the cooperation of the remote PE) than access to local memory†. For that reason we implement read requests to remote memory in a "split-phase" manner, where issuing the read request is separated from actually consuming the data [ARV87]. This allows an SP to issue the request and to continue executing the current SP

† Note that a system with a uniform memory access time would only simplify the problem since the distribution of data would be of much less concern.

until the requested data is actually needed, or to perform a context switch in the meantime. Presence bits are used to indicate whether a given memory location contains a valid data item.

Under these architectural assumptions, it is necessary to decide how to distribute both, data and the corresponding computation. Since arrays are the most important data structures in scientific computation, we concentrate on distributing *for-loops* operating over *arrays*. The main objective is to distribute computation evenly while keeping the number of remote data accesses to a minimum. PODS employs two techniques to achieve that. First, it uses the distribution of arrays to control the distribution of loop execution. This is called *Data-Distributed Execution*, and is accomplished in the following two conceptual steps:

1. Using a simple global algorithm, divide a given array into equal-size partitions and allocate each partition to a different PE.

2. Attempt to execute the loop iteration that *writes* a particular array element (there is only one under single assignment) on the same PE that holds that element.

The second technique is that of remote data *caching*. When a PE needs to read a remote data element, it send a message to the PE holding that element. This PE extracts the entire page containing that element and returns it to the requesting PE, where it is saved in a software cache. Due to locality of reference, this reduce the need for future remote requests to elements on the same page. The need is not completely eliminated because not all elements will, in general, be present at the time the page is transmitted. Hence the same page may be copied multiple times in the future as references to previously empty elements are being made. Note, however, that due to single assignment, there are no cache coherence problems and hence a cached page will never have to be sent back to the original owner.

The following sections present the data and program distribution mechanisms in detail.

4.1. Array Partitioning and Distribution

The ability of code execution to follow the distribution of data (step 2 above) depends greatly on how well the direction in which a matrix is accessed by the code (e.g., row-major vs column-major) matches the direction in which the matrix is cut up and distributed. This suggests that, instead of using a simple global algorithm for distributing all arrays, the compiler might attempt to distribute a given array based on an analysis of the code that accesses the array. Unfortunately, problems with aliasing (parameter binding) make a compile-time analysis very difficult. Furthermore, the same array may be used multiple times under different access patterns. Hence, instead of attempting to determine the best distribution pattern a priory, a better approach is to use the same pattern at all times, letting the programmer know which pattern is preferable. This is the approach used by many popular languages today. For example, 'C' uses row-major and FORTRAN uses column-major storage for 2-dimensional arrays. PODS uses row-major storage. The algorithm for distributing a given array is then as follows:

1. The array is cut-up row-major into pages of a fixed size, where the size is determined by the hardware architecture. For the iPSC/2, the best page size has been determined to be 32 elements or approximately 2 kilobytes. (Previous studies have shown that this is not a critical parameter [BIC89].)
2. Pages are grouped into segments of approximately equal size, which are assigned to PEs sequentially. The number of segments corresponds to the number of PEs.

To illustrate this partitioning, consider the following example. A two dimensional 6 x 256 array is to be partitioned and distributed over 4 PEs. There are 1536 elements in the array, resulting in 48 pages, i.e., 12 pages per PE. The diagram

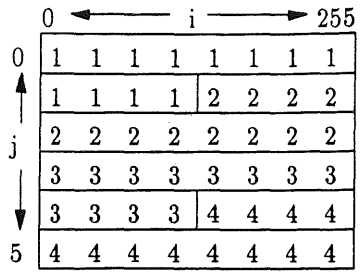

Figure 4

Partitioning of a 6 x 256 array over 4 PEs

in Figure 4 illustrates which pages are mapped onto which PE, where each of the digits (1, 2, 3, 4) represents one page. That is, PE1 holds the first 12 pages, PE2 holds the next 12 pages, and so on.

Each PE keeps track of its area of responsibility using an array header, built at the time the array is allocated. This contains the array dimensions and, for each dimension, the starting and ending indices. As will be explained in Section 4.2.2, this information is used by the range filter at run time to determine whether a given computation is to be performed locally.

The distribution of each array is performed at run-time. It is implemented using a special *distributing allocate* operator, which functions as follows:

1. The operator requests a new array ID from the local Array Manager (see Section 5.1).
2. When the Array Manger receives the allocate request, it allocates the necessary space, builds the array header, returns the array ID to the requesting SP, and then sends a remote allocation request to all other PEs with the array ID attached. In this way all PEs receive the same ID for the same array.
3. Each of the remote PEs receiving the allocate request builds the corresponding header and allocates the appropriate space.

Note that the SP initiating the allocation is not blocked while the allocate operation is in progress. Instead, it continues executing until it encounters an instruction that actually needs the array's ID as an operand. If the Array Manager has not yet responded by filling in that operand, the SP will block, causing a context switch; otherwise it continues executing.

4.2. Distributing Execution

As mentioned earlier, computation in PODS is distributed by following the Data Distributed Execution principle, which tries to map the calculation of an array element to the same PE that owns that element. This is achieved using the distributing L operator and the Range Filter, as described next.

4.2.1. The Distributing L Operator

The original Id dataflow graphs use the L operator to enter each new loop nest. The operator's function is to create a new context, thus distinguishing all data tokens belonging to the same loop instance.

In PODS, each loop nest corresponds to a separate SP. Hence each L operator transmits data tokens to a new SP, which is instantiated when its first instruction receives its operands.

To allow process distribution, we distinguish two forms of the L operator – the regular (local) L, and the new distributing L, called L^D. Both operate as stated above, i.e., they transmit tokens to another SP. The main distinction, however, is the location of the new SP. In the case of the local L operator, the values remain within the same PE and hence a *single* instance of the new SP is created locally. In the case of L^D, the same data value is replicated and routed to *all* PEs, thus causing an instance of an identical SP to be spawned on every PE. To cause each of the SPs to operate on a different data set, Range Filters are inserted into the SPs, as is described next.

4.2.2. Range Filters

The objective of the Range Filter construct is to control which iterations of a distributed loop are to be executed by a given PE. Conceptually, the Range Filter (RF) may be viewed as a predicate inserted into the dataflow graph. Its function is to discard index tokens that are not within the PE's range of responsibility.

The dashed arrows labeled "RF" in Figure 2 indicate the location where a Range Filter would be placed. In the case of the outer index, i, the RF simply discards all values that are outside its PE's area of responsibility, which is determined from the header of the array written by this loop. The RF for the inner loop performs a similar function for the j index. Note, however, that the legal ranges for j depend on i, which must be made available to the RF. For example, the RF in PE1 (see Figure 4) produces the j range 0:255 when i is 0 but only 0:127 when i is 1.

The RFs as explained so far are only conceptual. There is no need to produce the entire index range in each PE and then discard all of it except for a small local subrange. A more efficient way is to replace the entire index generation subgraph by a modified version, which generates *only* the desired subranges in each PE. For an ascending loop, these modifications are shown in Figure 5. The initial index value, init-i, is replaced by the *maximum* of init-i and the starting index of the PE's area of responsibility. Similarly, the test for the ending value, n, is replaced by the *minimum* of n and the ending index of the PE's area of responsibility. (In the case of a descending loop, the minimum and maximum operators are simply interchanged.)

For the inner loop, the initial and final values for j are extended in an analogous fashion. The only difference is that these values also depend on the current i. Hence the i-values must be fed into the maximum/minimum computations, as shown in the graph. The L^D operators are placed in front of the outermost loop that contains

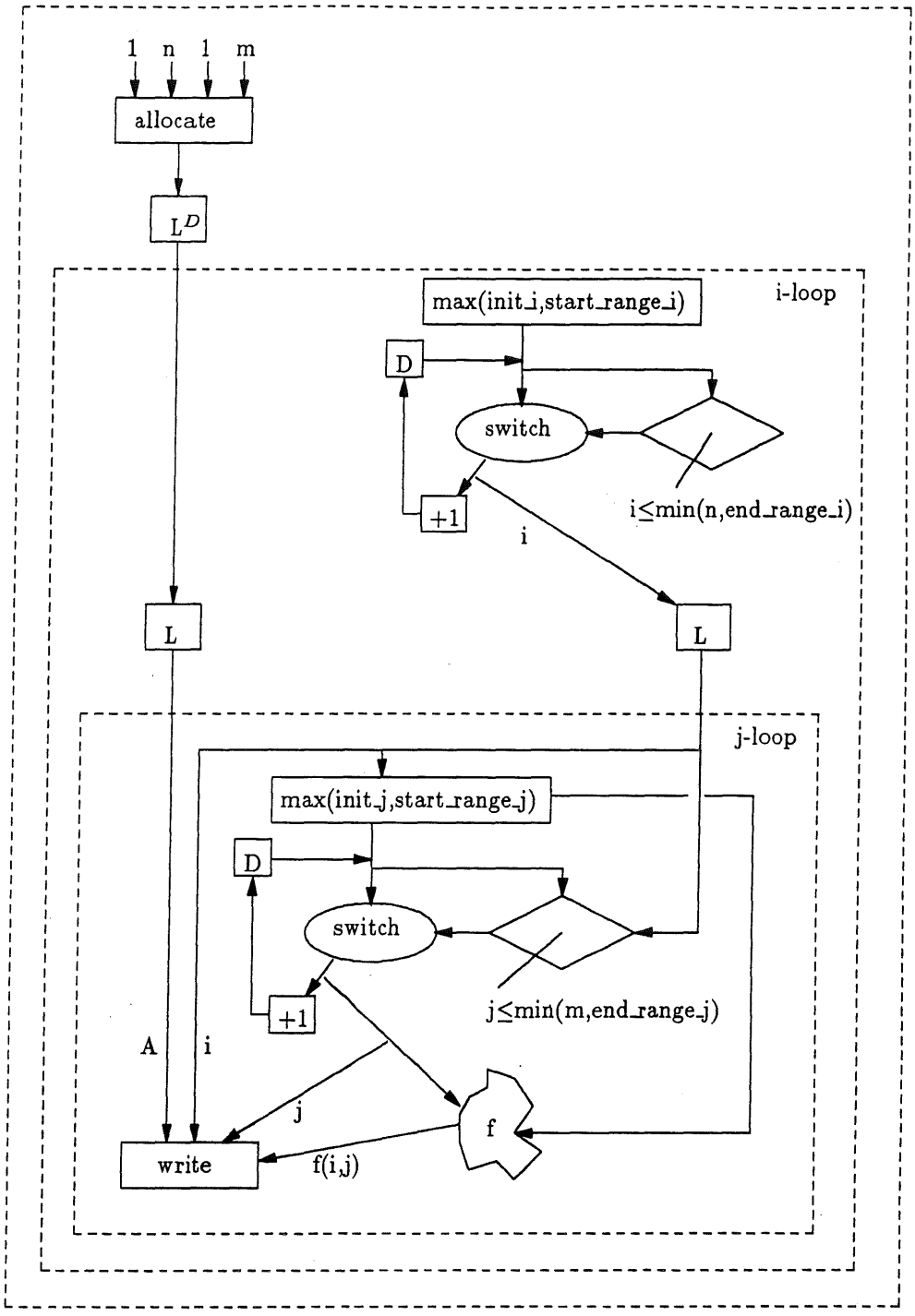

Figure 5

Modified Range Filters

a RF. In the above example, this would result in the SP corresponding to the i-loop to be replicated on all PEs, each operating on a different i subrange. For each iteration of this SP (i.e., for each i), a new SP corresponding to the j-loop is created and executed locally; each of these SPs operates on the j range corresponding to the given i.

4.2.3. RF Placement

In the previous discussion, we have assumed that each level of a nested loop would get its own RF. This section discusses two enhancements which have been implemented in order to reduce the number of RFs in nested loops. In the simulations described in Section 5, only one RF was used in any given loop, regardless of the level of nesting.

Consider an n-dimensional index space, where the dimensions are ordered by the levels of nesting. Say this multiple nested loop has index levels i_1, i_2, ..., i_n, with the SPs numbered correspondingly SP1, SP2, ..., SPn. It is possible to eliminate the RF at level i_1 by running only one instance of SP1. That is achieved by placing the L^D operator one level below the outer scope, i.e., into SP1 instead of its parent. Since there is only one instance of SP1, there is no need for a RF in that SP. However, all indices i_1 must now be broadcast to all PEs, where the RFs in the next-level SP2's check both i_1 and i_2 for the range boundaries. Eliminating the RF in SP1 is particularly appropriate when there is a loop-carried dependency at that level. In this case, the individual iterations within SP1 cannot proceed in parallel; at best, they will run in a staggered (doacross-like) manner. Hence distributing them over multiple PEs is not likely to improve performance.

This principle can be generalized to more than one level. That is, we can eliminate the RFs for all levels from i_1 to some level i_k by placing the L^D operator into SPk, thus causing SPk+1 to be distributed while SP1 through SPk remain

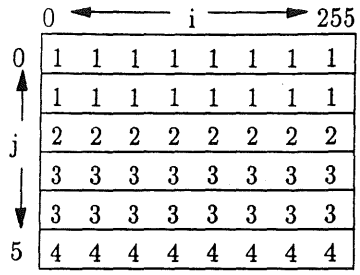

Figure 6

Index Space Partitioning

centralized. Again, we use the presence of loop-carried dependencies as a guide to select the level i_k.

A different technique can be used to eliminate RFs *below* a certain level ik. In this case, the RFs for SPk+1 through SPn are simply not included. This, however, does not come for free. The problem is that segment boundaries, in general, do not fall on array boundaries, as was illustrated for example in Figure 4. The consequence is that more than one PE may be responsible for the same index value. In the above example, both PE1 and PE2 own a portion of the row i=1 and hence both must receive the value 1 for i. If there is no RF at the level j to discriminate between the different subranges at that level, both PEs would generate all values for j and hence the same row i=1 would be computed twice.

One possible solutions to this problem is to assign the conflicting row to only one PE. We have implemented a simple rule to decide on the responsibility: the PE holding the *first* element of any given row is responsible for the entire row. The necessary consequence is that some number of remote writes will occur, since the index space partitioning does not exactly follow the partitioning of the array.

To visualize this approach, consider the diagram in Figure 6. This shows the distribution of the iteration space for a nested loop operating on the array in Figure 4. In particular, PE1 is responsible for the first two rows, even though it

only holds the first half of the second row in its local memory. PE2, on the other hand, computes only row 2; the values for the second half of row 1 are are sent to it by PE1.

Note that once the RF below a given level i_k is eliminated, the RFs at all lower levels become superfluous. This is because the RF at level i_k partitions the index space along the i_k dimension into disjoint subranges (they are disjoint because of the first-element-ownership rule discussed above). For all levels below i_k the index ranges are then needed in their entirety (for every element of i_k).

4.2.4. For-Loop Distribution Algorithm

By combining the two techniques described in Section 4.2.3, we can eliminate all RFs but one for any given nested loop of any depth. The following is the actual distribution algorithm used currently by PODS to distribute data and program execution:

1. Given an array A, partition and distribute it as described in Section 4.1.
2. Starting with the outer-most code block, repeat the following until all sets of nested loops are marked (depth-first traversal) as either distributed or local.
   a. Consider the next inner code block. If this code block does not have a loop-carried dependency (LCD), then mark it; all descendent SPs will be local.
   b. If this inner SPs has an LCD, then goto step 2.
   c. If this is the inner most SPs, then consider the next unmarked SPs (depth-first) and goto step 2.
3. In each marked SP replace the predicate with a Range Filter.
4. In the parent of each marked SP change the L operators into L^D operators.

It is important to point out that the detection of LCDs in a declarative language is considerably simplified due to its side-effect free semantics and the

lack of general pointers to do aliasing. Furthermore, the only possible form of dependency is flow dependency. Despite these favorable characteristics, there are always cases where the presence or absence of LCDs cannot be determined at compile time. This, however, is not a significant limitation in our case. If the compiler fails to detect an existing dependency, the single assignment principle still guarantees a deterministic program behavior irrespective of timing issues. Hence, contrary to conventional language compilers, the detection of LCDs is only a useful heuristic and not a necessity to determine whether a loop can be distributed.

5. Simulation

5.1. The Target Architecture

The target architecture for PODS is a MIMD architecture with distributed memory. Withing each PE, there are multiple functional units, each dedicated to a specific function. Even though the ultimate goal of the project is to implement PODS on iPSC/2, where each PE consists of an Execution Unit (ALU), local memory, and a routing unit, we have simulated all functional units as if they were separate hardware units operating concurrently. The main reason for this was to find out which functional unit would be the most critical and thus would need the most efficient implementation. As will be discussed shortly, most of the units other than the actual execution unit were only lightly loaded most of the time and hence can easily be implemented within the existing iPSC node.

Figure 7 shows the overall organization of a PE. This architecture was simulated at the instruction level. In order to compare the results of PODS simulations to the outside world, the Simulator is set-up as if it were executing on Intel 386 microprocessors in a hypercube configuration. These are Intel 80386/80387 CPU's at 16 MHz with Direct-Connect Modules for communication. Each of the tasks and the timing assumptions of the functional units is explained below.

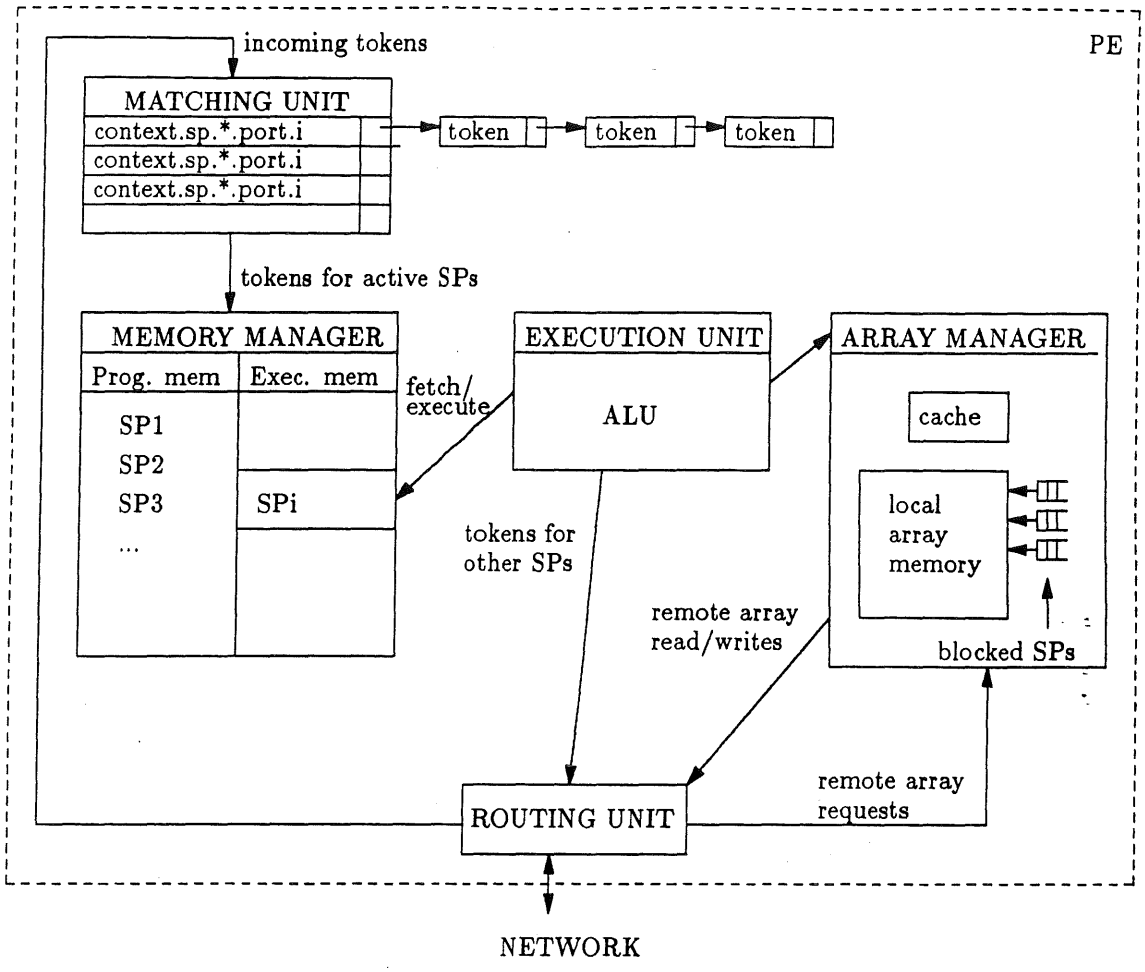

Figure 7

Logical Units of a PODS PE

Matching Unit

When an input token arrives (via the Routing Unit), it is run through the Matching Unit. If the corresponding SP is already active, the token is passed to it immediately. Otherwise it is enqueued at the SP's entry in the Matching Unit. This is implemented as a hash table lookup based upon the SP ID, and the frame pointer. It takes 15 μseconds.

Note that only a small percentage of all tokens generated during a computation actually pass through the Matching Unit. Most tokens are produced and

consumed internally within the same SP and hence are stored directly into the appropriate operand slots. Only tokens exchanged between different SPs go through the Matching Unit.

Memory Manager

The Memory Manager maintains two separate memory area: a Program Memory, where the code for all SPs is kept, and an Execution Memory, containing all currently active SPs. The Memory Manager loads an SP from Program Memory into Execution Memory when the first instruction of that SP is enabled, and it releases the space when the SP terminates. To perform these tasks, it must allocate/deallocate execution memory frames from free memory, which are maintained as linked lists. We assume that each add or delete operation from the linked list takes approximately 3 memory references or 0.9 μseconds.

Execution Unit

The Execution Unit is a conventional von Neumann ALU which executes the current SP in a control-driven manner, using a simple program counter. A context switch occurs when a disabled instruction is encountered. Most of the time, data tokens are produced and consumed within the current SP, i.e., they are written into and read from appropriate slots in Execution Memory. The only exceptions are array accesses, which are passed on to the Array Manager, and tokens destined for other SPs, which are passed on to the Routing Unit.

The timing of the Execution Unit is based upon three calculations: (1) the time of each normal (local) operation; (2) the time it takes to perform a fast context switch; and (3) the time to perform a local array read (The time for remote accesses and for token routing is accounted for by the Array Manager and the Routing Unit, respectively.)

The time for each normal operation was measured on the iPSC/2 with the following results:

iPSC/2 Instruction	Execution time (microsec)
integer add	0.300
integer subtraction	0.300
bitwise logical	0.558
floating point negate	0.555
floating point compare	5.803
floating point power	96.418
floating point abs	12.626
floating point square root	18.929
floating point multiply	7.217
floating point division	10.707
floating point addition	6.753
floating point subtraction	6.757

The time for a local array read is based on the following pseudo code:

```
offset = size_dim2 * i + j
if (offset < beginning_offset) goto REMOTE_READ
if (offset ≥ ending_offset) goto REMOTE_READ
if (element not present) goto ENQUEUE_READ
value = array[offset]
```

The time for a local array read (assuming the value is present) is: 1 integer multiply + 1 integer add + 3 integer comparisons + 1 local read. This works out to be 2.7 μseconds.

The time for a fast context switch is based on the 80386 CALL ptr16:32 instruction. This is a full 32 bit indirect procedure call. The worst case for this is 21 clock cycles or 1.312 μseconds at 16 Mhz.

Routing Unit

This unit is responsible for taking a token, forming a message, and sending it over the network to the correct PE and SP. It is also responsible for receiving array read/write requests from other PEs, which it passes on to the Array Manager. Dunigan [DUN88] has done extensive testing of the iPSC/2 and found that the communication can effectively be expressed using the following equations:

if (message_length $\leq$ 100 bytes) then 390 microsec
if (message_length $>$ 100 bytes) then 697 + 0.4 * message_length μsec

When the Routing Unit receives a token to route, a simple table look-up is used to find the destination SPs. This is then used in a hash function to find the destination PE. Since tokens are less than 100 bytes, and they are batched together in groups of 20, the simulation uses an estimate of 19.5 μseconds for each token added to a batch.

Array Manager

The Array Manager handles all array allocations and array accesses. To allocate an array, the Array Manager on the PE where the allocate operator is initiated assigns the array a unique ID and broadcasts the request to all PEs as described in Section 4.1.

To perform a local array read, the Array Manager determines whether the element is present or absent. In the first case, the value is simply read and returned to the Execution Unit. In the second case, the request is enqueued by setting a flag in the memory location of the cell to indicate that there are requests which will need to be serviced when the cell is written. This is much like the implementation of I-structures [ARV89, ARV87].

To perform a remote array read, the Array Manager first examines the cache. If the element is present, it is read just like a local element. Otherwise, a request is sent via the Routing Unit to the appropriate PE. If the value is present in the target PE then the entire page is returned and cached in the requesting PE; if it is absent, the request is queued in the target PE.

To perform a write, the Array Manager also distinguishes between a local and a remote location. (Due to single assignment, the value cannot yet exist in the cache.) If the location is local, the value is written into that location and the SPs blocked on that location are reactivated. For a remote location, the value is sent to

the target PE, which writes it into the appropriate array slot and also reactivates all PEs blocked on that location.

As mentioned earlier, single assignment guarantees that there are no cache coherence problems. Hence a write operation need not be propagated to other PEs, since that element cannot exists in any PE's cache.

The Array Manager handles the following tasks in the indicated times:

Free Array: array_size * memory_read_time
Array Write: memory_write_time + number_queued_reads * message_time
Cached Read: memory_read_time + message_time if not present
Remote Read: memory_read_time + enqueued_read_time or message_time
Receive Page: page_size * memory_write_time
Send Page: page_size * memory_read_time + message_time
Allocate Array: 100.0 seconds + message_time
where
memory_read_time is the time for a local read = 0.3 μsec
memory_write_time is the time for a local write = 0.4 μsec
message_time is the time for a signal from one functional unit to another on the same PE = 1.0 μsec
enqueued_read_time is the time to push an early read onto a stack = 3 * memory_read_time + 5 * memory_write_time = 2.9 μsec

Network

Since the Routing Unit handles all of the transmission setup, the Network models only the physical propagation time. The iPSC/2 has a theoretical 100 Mbyte per second bandwidth. Assuming each message is approximately 100 bytes, the time for 1 hop is 1 μsecond. The network time is set to 2.5 μseconds, simulating an average of 2.5 hops.

5.2. SIMPLE

In addition to running a few generic examples, such as matrix multiply, we have concentrated on the SIMPLE benchmark [Cro78], developed by Lawrence Livermore Laboratory. This code is a hydrodynamics and heat conduction simulation and is indicative of the large-scale scientific code which is executed on

supercomputers today. It simulates the behavior of a fluid in a sphere, using a Lagrangian formulation and equations.

SIMPLE consists of three major routines: velocity_position, hydrodynamics, and conduction. All of the other procedures are either run only once or are called by one of the above. Each routine is essentially a set of deeply-nested loops operating on multi-dimensional arrays. The most important routine is conduction; both velocity_position, and hydrodynamics are much easier to parallelize. Velocity_position has no LCDs, no function calls, and runs in parallel very well. Hydrodynamics has only 5 SPs and is basically one big nested loop. Conduction is the most difficult to parallelize because of: (1) the sweep phases where every element is recalculated twice, based upon its neighbors; (2) the complexity of the resulting 15 SPs plus multiple function calls; and (3) the large number of LCDs with both ascending and descending for-loops. These LCD's make iteration level parallelism a challenging task.

The following sections give the results of executing SIMPLE under the conditions described in Section 5.1. The program, written in ID Nouveau, was first translated by the MIT compiler. The resulting dataflow graphs were then transformed automatically into distributed SPs, as indicated in Figure 3. No optimization techniques, except for standard scalar expansion, were applied.

5.3. Results

5.3.1. Functional Unit Balance

This addresses the question of balance among the various functional units within a PE and is measured as the fraction of the time a given facility is busy. Figure 8 summarizes the results for a 16 x 16 problem size. This shows clearly that the Execution Unit (EU) is the most heavily utilized. The most important implication of these measurements is that there is no need for any specialized

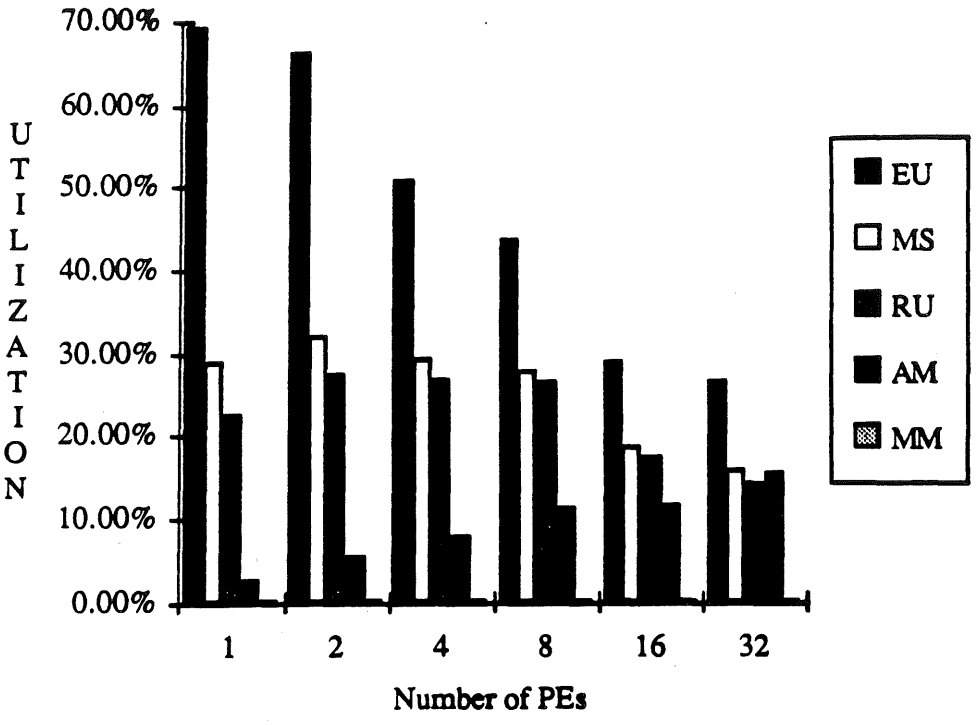

Figure 8

Average Utilization of Each Functional Unit

hardware units to support the system. The supporting functional units can all be implemented in software, running on the same iPSC processor as the Execution Unit.

5.3.2. Average Execution Unit Utilization

Having established that the Execution Unit is the most critical unit in the system, we now investigate its utilization in more detail.

Figure 9 shows the results for different problem sizes. For a 64 x 64 SIMPLE the utilization starts out at approximately 70% for 1 PE and goes down to 50% for 32 PEs. On smaller problems (16 x 16) the Execution Unit utilization is lower than on large problems, especially when the number of PEs is large. It is, however,

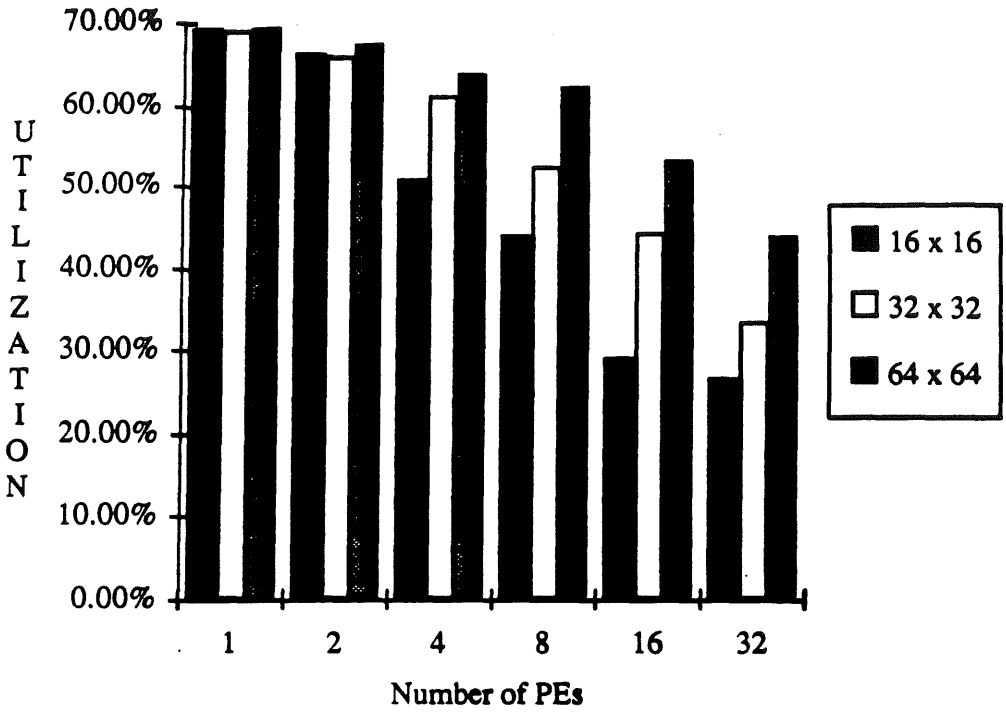

Figure 9

Execution Unit Utilization for SIMPLE.

interesting to note that SIMPLE continues to speed-up even when the Execution Units are 50% idle (see Figure 10 below).

5.3.3. Scalability

This measures how much a problem speeds-up as the number of PEs is increased, and is perhaps the single most important characteristic of a multiprocessor system. Speed-up is defined to be the time of a single PE run divided by the time of the multiple PE run. Figure 10 shows the speed-up for different problem sizes. The 45° curve represents ideal speed-up (100%). For comparison the speed-up obtained by Pingali and Rogers [PIN90] for a 64 x 64 run is also plotted (P&R).

For the small 16 x 16 case, PODS tops out at a speed-up of 8.1. For the 32 x 32 case, speed-up tops out at 12.4, i.e., more than an order of magnitude. The 64

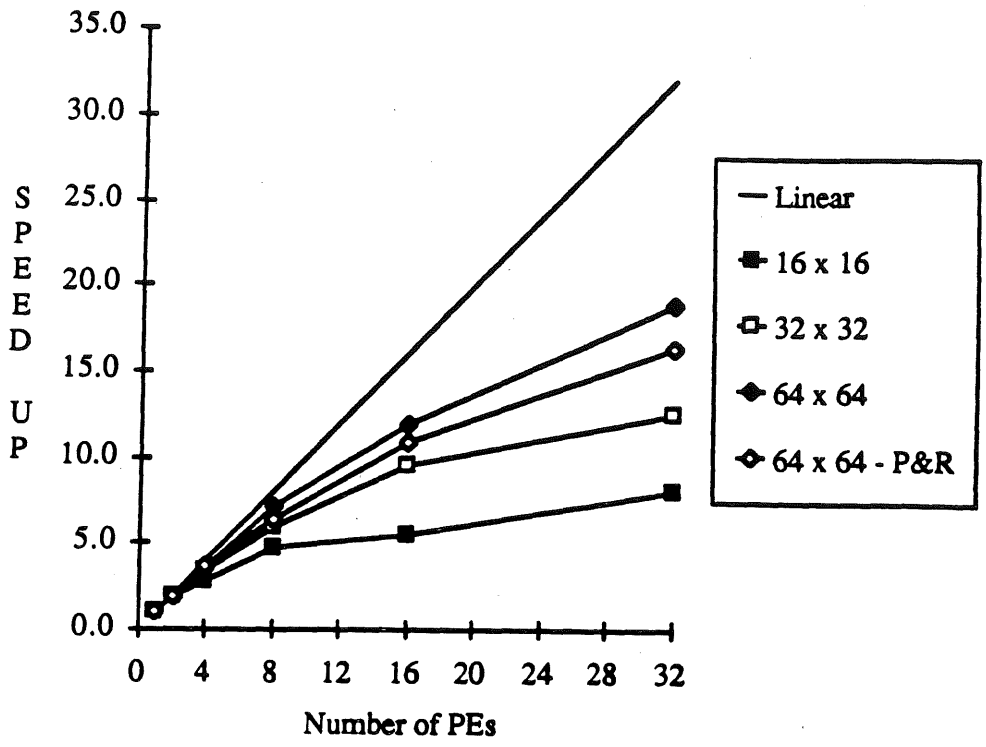

Figure 10

Speed-Up of SIMPLE

x 64 problem size is much more typical of a "real" hydrodynamics simulation and is thus a better gauge for the success of PODS in parallelizing scientific code. For the 64 x 64 case, PODS is able to spread the work efficiently across all of the PEs, achieving a speed-up of 18.9 on 32 PEs.

5.3.4. Efficiency Comparison

When studying speed-up, it is important to consider the efficiency of the parallel version running on a single PE as compared to the most efficient sequential version (written in a conventional language). Typically, the parallel version will be less efficient because of the additional tasks that must be performed for multiple PEs even though there is only one operating. Also, commercial systems provide a variety of additional optimizations which research systems may not offer. Only

if this comparison shows that the parallel system running on one PE is within some reasonable percentage of the sequential version, can the scalability results be considered to have a valid base time.

We have compiled a sequential version of SIMPLE, written in C, using the Intel-supplied compiler, and timed its execution on the iPSC/2 host. A 32 x 32 input conduction takes 0.9 seconds on a single iPSC/2 PE. The PODS Simulator estimated that the program would run in 1.72 seconds. This is approximately twice the time of the commercial version, and shows that PODS, when running sequentially, is not grossly inefficient. This has been found to be true of all the test cases, thus giving credence to the scalability results presented in Section 5.3.3.

6. Conclusions and Comparison with Related Work

The objective of this project is to demonstrate that declarative programming is a suitable approach to the programming of conventional coarse-grain multiprocessors. This objective is similar to that of Pingali and Rogers [PIN90, ROG89]; the approaches, however are quite different. Their approach is based on compiling Id programs into C for execution on the iPSC/2. Once the programs are compiled into native code, processes are statically scheduled onto processor nodes and execution proceeds in a completely control-driven manner. With PODS, execution is still driven by the production and availability of data. First, an SP is instantiated by the arrival of its operands; furthermore, while its progress is governed by a program counter, the availability of operands governs the state transitions between the ready, blocked, and running states. As shown in Figure 10, PODS outperformed the pure compilation approach using the SIMPLE benchmark when the problem size was sufficiently large.

While PODS is closest to the above compilation approach in its *objective*, namely to utilize a commercially available conventional multiprocessor, in its *approach* it is closest to a dataflow hybrid, a number of which have beed developed in the recent past [CUL91, GAO90, GAU90, GRA89, IAN88, NIK89, ROG89, SAK89]. What these approaches have in common is the idea of creating sequential processes or threads out of dataflow programs. In most cases, the threads are very short, comprising at most a few dozens of instructions. To compensate for the overhead associated with the frequent context switching, special-purpose architectures (processors) are necessary. Hence the term "hybrid" refers to the fact that the underlying architecture combines the features of both von Neumann and dataflow computers.

The objective in PODS, on the other hand, is to use standard von Neumann processors interconnected into a multi-computer or multi-processor architecture with long remote memory latencies. Consequently, the process granularity is much larger, comprising entire ranges of loop iterations and function invocations. As has been demonstrated with the SIMPLE benchmark, there are sufficient amounts of iteration-level parallelism in scientific code to adequately exploit the hardware parallelism of medium-scale multiprocessors, without having to expose parallelism at a finer level.

The basic mechanisms that work together to achieve iteration-level parallelism in PODS are the distributing allocate operator for subdividing arrays over multiple PEs, the distributing L operator for spawning multiple instances of the same process on multiple PEs, and the Range Filter, which divides the index space of a loop such that each process operates on a different subrange. We wish to point out that these mechanisms, while developed and presented in the context of Id, are not restricted to Id or to a declarative language in general. They rely primarily on the single-assignment principle to guarantee deterministic program

execution, and thus could be incorporated into any language that supports this style of programming.

References

[Ack79] Ackerman, W.B., Dennis, J. Val – A Value-Oriented Algorithmic Language. *Technical Report MIT/LCS/TM-218* (June, 1979), Laboratory for Computer Science, Massachusetts Institute of Technology.

[Arv86a] Arvind, Culler, D.E. Dataflow Architectures. *Technical Report MIT/LCS/TM-294* (February, 1986), Laboratory for Computer Science, Massachusetts Institute of Technology.

[Arv87] Arvind, Nikhil, R.S. Executing a Program on the MIT Tagged-Token Dataflow Architecture. *Computation Structures Group Memo 271* (March, 1987), Laboratory for Computer Science, MIT, Cambridge, MA.

[Arv87a] Arvind, R. S. Nikhil, K. K. Pingali ID Nouveau Reference Manual Part II: Operational Semantics.. *Technical Report* (April, 1987), Laboratory for Computer Science, Massachusetts Institute of Technology.

[Arv88] Arvind, Ekanadham, K. Future Scientific Programming on Parallel Machines. *J. Parallel Dist. Comp. V5*, n5 (1988), 460-493.

[Arv89] Arvind, R. S. Nikhil, K. K. Pingali I-Structures: Data Structures for Parallel Computing. *ACM TOPLAS V11*, n4 (1989), 598-632.

[Bac78] Backus, J. Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs. *Communications of the ACM 21* (August, 1978), 613–640.

[Bic90] Bic, L. A Process-Oriented Model for Efficient Execution of Dataflow Programs. *Jour. Parallel and Distr. Computing 8* (1990), 42-51.

[Bic89] L. Bic, M. D. Nagel, J. M. A. Roy Automatic Data/Program Partitioning Using the Single Assignment Principle. *Supercomputing '89* (1989), 551-556.

[Cro78] W. P. Crowley, C. P. Henderson, T. E. Rudy The SIMPLE Code. *UCID 17715* (February, 1978), Lawrence Livermore Laboratory..

[Cul89] Culler, D.E. Managing Parallelism and Resources in Scientific Dataflow Programs. *PhD Thesis* (1989), Laboratory for Computer Science, M.I.T., Cambridge, MA.

[Cul91] Culler, D.E., et al Fine-Grain Parallelism with Minimal Hardware Support: A Compiler-Controlled Threaded Abstract Machine. *Proc. ASPLOS-IV* (April,, 1991), Santa Clara, CA.

[DUN88] T. H. DUNIGAN Performance of a Second Generation Hypercube. *Technical Report ORNL/TM-10881* (November, 1988), Oak Ridge National Laboratory.

[GAU90] GAUDIOT, J. L., EVRIPIDOU, P. The USC Decoupled Multilevel Data-Flow Execution Model. In *Advanced Topics in Data-Flow Computing*, J-L. Gaudiot, L. Bic, Ed., Prentice-Hall, Inc., 1990.

[GAO90] GAO, G. R. A Flexible Architecture Model for Hybrid Dataflow and Control-Flow Evaluation. In *Advanced Topics in Data-Flow Computing*, J-L. Gaudiot, L. Bic, Ed., Prentice-Hall, Inc., 1990.

[GRA89] GRAFE, V.G., DAVIDSON, G.S., HOCH, J.E., HOLMES, V.P. The Epsilon Dataflow Processor. *Proc. 16th Annual Int'l Symp. on Computer Arch.* (1989).

[IAN88] IANNUCCI, ROBERT A. Toward a Dataflow/von Neumann Hybrid Architecture. *Proc. 15th Int'l Symp. on Computer Architecture* (1988).

[LAN65] P.J. LANDIN A Correspondence Between ALGOL 60 and Church's Lambda-Notation: Part I. *Comm. ACM V8*, n2 (1965), 89-101.

[MCG85] MCGRAW, J.R., SKEDZIELEWSKI, S., ALLAN, A., GRIT, D., OLDEHOEFT, R., GLAUERT, J.R.W., DOBES, I., HOHENSEE, P. SISAL—Streams and Iterations in a Single-Assignment Language. *Language Reference Manual, TR M-146* (March, 1985), Lawrence Livermore Lab..

[NIK87] R.S. NIKHIL ID Nouveau Reference Manual Part I: Syntax. *MIT Technical Report* (April, 1987), Laboratory for Computer Science, MIT.

[NIK88] R.S. NIKHIL ID Reference Manual – Version 88.1. *Computation Structures Group Memo 284* (August, 1988), Laboratory for Computer Science, MIT, Cambridge, MA.

[NIK89] R.S. NIKHIL, ARVIND Can Dataflow Subsume von Neumann Computing?. *16th Int'l Computer Architecture Conference, Jerusalem* (1989), 262-272.

[PIN90] K. PINGALI, A. ROGERS Compiler Parallelization of SIMPLE for a Distributed Memory Machine. *TR 90-1084* (January, 1990), Department of Computer Science, Cornell University.

[ROG89] A. ROGERS, K. PINGALI Compiling Programs for Distributed Memory Architectures. *4th Hypercube Concurrent Computers and Applications Conference* (1989), 529-542.

[SAK89] SAKAI, S., YAMAGUCHI, Y., HIRAKI, K., KODAMA, Y., TUBA, T. An Architecture of a Dataflow Chip Processor. *Proc. 16th Annual Int'l Symp. on Computer Arch.* (June, 1989), Jerusalem.